To appear in *Optimization Methods & Software*
Vol. 00, No. 00, Month 20XX, 1–17

Extending the Binomial Checkpointing Technique for Resilience

Andrea Walther[a*] and Sri Hari Krishna Narayanan[b]

[a]*Institut für Mathematik, Universität Paderborn, Paderborn, Germany;* [b]*Mathematics and Computer Science Devision, Argonne National Laboratory, Lemont, USA*

(*December 23, 2016*)

In terms of computing time, adjoint methods offer an attractive alternative for computing gradient information, required, for example, for optimization. Together with this very favorable temporal complexity result, however, comes a memory requirement that is in essence proportional to the operation count of the underlying function, for example, if algorithmic differentiation is used to provide the adjoints. For this reason, checkpointing approaches in many variants have become popular. This paper analyzes an extension of the so-called binomial approach to cover also possible failures of the computing systems. Such a measure of precaution is of special interest for massive parallel simulations and adjoint calculations where the mean time between failure of the large-scale computing system is smaller than the time needed to complete the calculation of the adjoint information. We describe the extensions of standard checkpointing approaches required for such resilience, provide a corresponding implementation and discuss numerical results.

Keywords: Binomial Checkpoining, Resilience, Computation of Adjoints

1. Introduction

The use of adjoint methods allows the computation of gradient information within a time that is only a small multiple of the time needed to evaluate the underlying function itself. As soon as the considered process is nonlinear, however, the memory requirement to compute the adjoint information is in principle proportional to the operation count of the underlying function, see, for example, [6, Sec. 4.6]. In Chap. 12 of the same book, several checkpointing options to reduce this high memory complexity are discussed. Checkpointing strategies use a small number of memory units (checkpoints) to store the system state at distinct times. Subsequently, the information that is needed for the adjoint computation but is not available is recomputed by using these checkpoints. Several checkpointing techniques have been developed, all of which seek an acceptable compromise between memory requirement and runtime increase.

For this paper, we assume that the evaluation of the function of interest has a time-step structure given by

$$x_i = F_i(x_{i-1}, u_{i-1}), \quad i = 1, \ldots, l \,, \tag{1}$$

for a given x_0, where $x_i \in \mathbb{R}^n$, $i = 0, \ldots, l$, denote the state of the considered system and

*Corresponding author. Email: andrea.walther@uni-paderborn.de

$u_i \in \mathbb{R}^m$ the control. The operator $F_i : \mathbb{R}^n \times \mathbb{R}^m \mapsto \mathbb{R}^n$ defines the time step to compute the state x_i. The process to compute x_l for a given x_0 is also called forward integration. In order to optimize a specific criterion or to obtain a desired state, the cost functional

$$J(x(u), u) = J(x, u)$$

measures the quality of $x(u) = (x_1, \dots, x_l)$ and $u = (u_1, \dots, u_l)$, where $x(u)$ depends uniquely on the control u if the involved functions are smooth enough. For applying a derivative-based optimization method, one could use an adjoint integration of the form

$$\begin{aligned} \bar{u}_l = 0\,, \quad \bar{x}_l \quad \text{given} \\ (\bar{x}_{i-1}, \bar{u}_{i-1}) = \bar{F}_i(\bar{x}_i, \bar{u}_i, x_{i-1}, u_{i-1}), \quad i = l, \dots, 1, \end{aligned} \tag{2}$$

where the operator $\bar{F}_i$ denotes the adjoint time step. Subsequently to or concurrently with the adjoint integration, the desired derivative information $J_u(x(u), u)$ can be reconstructed from $\bar{x}$. As can be seen, the information of the forward integration (1) is needed for the adjoint computation (2). To provide this information within only a limited amount of memory, we use the binomial checkpointing approach proposed in [4, 5] as a basis to develop a checkpointing approach that can also handle a failure of the computing system. Such techniques include a foreseen suspension, where the application should suspend itself gracefully after completing the set number of forward or adjoint time steps. However, an unforeseen failure also has to be covered, where the application is killed externally because of machine failure or expired time allocation. This scenario occurs when running the MIT General Circulation Model (MITgcm, [1, 2]) on ARCHER, a UK-based supercomputer. MITgcm executes for around 351,000 time steps to simulate one year of physical time. This requires around 24 hours of wall-clock computation time. Because the mean time between failure of ARCHER is less than 24 hours, administrative policies that require applications execute for a fixed time allocation T, such as 6 hours at a time, before they are suspended. Typically, an application can be restarted from suspension when checkpoints containing intermediate data are available and the application is aware of its position in the overall computation.

An additional aspect that has to be taken into account is the actual location where checkpoints are stored. Checkpoints stored in memory can be lost on failure. For the sake of resilience or because future supercomputers may be memory constrained, checkpoints may have to be stored to disk. Therefore, the access time to read or write a checkpoint is not negligible, in contrast to the assumption frequently made for the development of checkpointing approaches. A few researchers have extended checkpointing techniques to a hierarchical checkpointing; see, for example, [3, 8, 9]. To derive a checkpointing technique that incorporates resilience, however, we ignore this hierarchical nature and assume throughout that the writing or reading process for a checkpoint is performed asynchronously such that it does not interfere with the adjoint computation.

This paper has the following structure. In Section 2, we describe binomial checkpointing and its implementation in a software system called revolve. Section 3 presents resilient binomial checkpointing, which is the adaptation of binomial checkpointing to cover resilience. The implementation of this approach, called resrevolve is also discussed. Theoretical results with respect to the temporal complexity of the resilient binomial checkpointing are given in Section 4. In Section 5 we discuss other checkpoint approaches for resilience. In Section 6 we draw conclusions and briefly discuss future work.

```
...
cp_schedule=new Schedule(l,c)
do
    whatodo = cp_schedule–>next()
    switch(whatodo)
        case advance: for cp_schedule–>oldcapo < i ≤ cp_schedule–>capo
                          forward(x, u)
        case takeshot: store(x, xstore, cp_schedule–>curr_checkpoint)
        case firsturn:  eval_J(x, u)
                        init(bar_u, bar_x)
                        adjoint(bar_x, bar_u, x, u)
        case youturn:  adjoint(bar_x, bar_u, x, u)
        case restore:   restore(x, xstore, cp_schedule–>curr_checkpoint)
while(whatodo <> terminate)
...
```

Figure 1. Implementation of the binomial checkpointing algorithm with calls to the revolve routines

2. Binomial Checkpointing Using revolve

For large-scale applications such as the MITgcm, only a limited number of intermediate states can be stored as checkpoints because of limited storage or the non-negligible time required to store intermediate states. Hence, one obvious question is where to place these checkpoints during the forward integration in order to minimize the amount of required recomputations. It was shown in [5] that a checkpointing scheme based on binomial coefficients yields for a given number of checkpoints the minimal number of time steps to be recomputed. The implementation of this optimal binomial checkpointing strategy is a software routine called revolve, that provides a data structure schedule to steer the checkpointing process and the storage of all information required for this purpose. Then, the forward integration as well as the corresponding adjoint computation is performed within a do-while loop of the structure in Fig. 1, where l denotes the number l of time steps of the forward simulation and and c the number c of checkpoints.

Hence, the routine revolve determines the next action to be performed that must be supported by the application being differentiated. For example, this action may be the execution of a part of the forward integration based on the routine forward(x,u), where x represents the state of the system and u the control, one step of the actual adjoint computation performed in the routine adjoint(bar_x,bar_u,x,u), where bar_x denotes the adjoint state and bar_u the adjoint control. For the checkpointing approach, the current state must be stored as a checkpoint; that is, execute the routine store(x, xstore, cp_schedule–>curr_checkpoint), where xstore represents an array to store system states and cp_schedule–>curr_checkpoint is the number of the entry, where the checkpoint has to be stored. In order to recompute the required intermediate information a checkpoint has to be read, that is, restore(x, xstore, cp_schedule–>curr_checkpoint) has to be performed.

We note that this checkpointing approach is completely independent of the method that is actually used to provide the adjoint information. As can be seen, as soon as an adjoint computation is available, only a limited effort is needed to combine this with binomial checkpointing in order to reduce the memory requirement. We stress that revolve provides so-called serial checkpointing, which means that only one forward time step or one adjoint step is performed at each stage of the adjoint computation. This is in contrast to so-called parallel checkpointing techniques where several forward time steps might be performed in parallel even in conjunction with one adjoint step.

Up to now we have assumed that for the serial checkpointing the computation of the forward time step and the adjoint step are free of failures. In reality, the computation of the forward step or the adjoint step may be performed heavily in parallel, that is, it may be evaluated on a large-scale computer system. This is precisely the situation where we have to take resilience into account and therefore an appropriately adapted extension of binomial checkpointing is required.

3. Binomial Checkpointing for Resilience

The ability to recover from a possible failure poses two additional challenges for the checkpointing scheme. First, the distance between two checkpoints should not be so large that a restart of the computation is too costly. Hence, resilient binomial checkpointing bounds the distance of two checkpoints in terms of the number of time steps that are performed before the next checkpoint is set. Second, since a failure may also occur during the adjoint computation, the adjoint state has to be checkpointed as well. Therefore, from now on we will distinguish between state checkpoint and adjoint checkpoints. Because of the nature of the adjoint computation, only one adjoint state is required to restart the adjoint computation. Since the adjoint state itself is assumed to be large, however, it is not possible to checkpoint every adjoint state computed. Therefore, resilient binomial checkpointing defines a distance between these so-called adjoint checkpoints. This distance corresponds to the number of adjoint steps performed, after which the current adjoint state is stored again. The resilient binomial checkpointing approach implemented in resrevolve uses resilience_distance for storing the maximal number of forward time steps between two checkpoints and adjoint_distance for storing the number of adjoint steps performed before the current adjoint state is stored again.

Internally, only the maximal distance between two state checkpoints, i.e., the value of resilience_distance interferes with the optimal binomial state checkpointing approach. To limit the number of time steps between two consecutive checkpoints, first the distance required for the optimal, i.e., binomial checkpointing, is computed. This number is then compared with the value of resilience_distance. If the value of resilience_distance is smaller than the number of time steps chosen by optimal binomial checkpointing, only resilience_distance steps are performed despite the fact that this might lead to a suboptimal checkpointing schedule. This comparison is performed each time a state checkpoint has to be stored. Therefore, resrevolve implements the optimal checkpointing approach whenever possible. An analysis of this possibly suboptimal checkpointing approach is presented in Sect. 4.

If a failure actually occurs, one faces two possibilities. First, the failure may happen during the forward integration. Then, the computation can just restart at the last checkpoint stored. Second, the failure may happen after the start of the adjoint computation. Then the checkpoint distribution at the time of the failure may differ from the one when the last adjoint checkpoint was written. As a small example we consider the case when $l = 100$ time steps are performed during the forward integration, five state checkpoints can be stored, the resilience distance is 30 and an adjoint checkpoint is stored every 12 adjoint steps. With standard binomial checkpointing, the following states would serve as checkpoints during the first forward integration of the state:

Checkpoint distribution at first adjoint step:	0	45	70	86	95

as illustrated also in Fig. 2. As can be seen, there are more than 30 time steps between

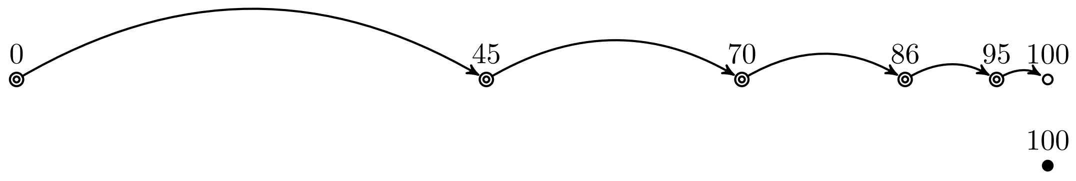


Figure 2. Checkpoint distribution at first adjoint step using regular revolve

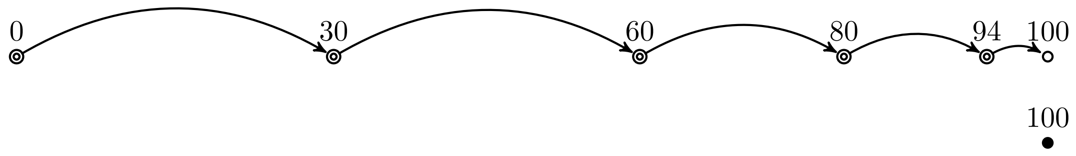


Figure 3. Checkpoint distribution at first adjoint step using resilient revolve

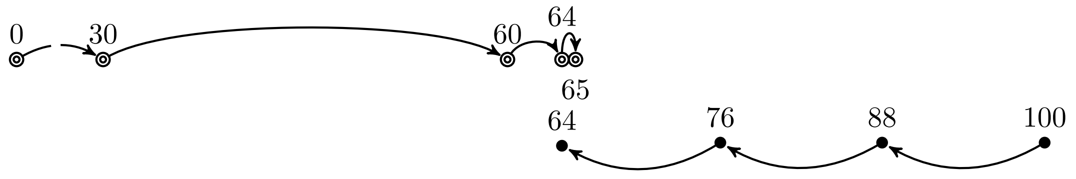


Figure 4. Checkpoint distribution at 64th adjoint step

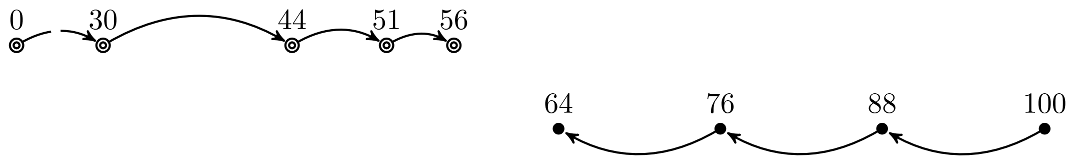


Figure 5. Checkpoint distribution at breakdown

the first and the second state checkpoint, violating the bound of 30 for the resilience distance. Using the resilient binomial checkpointing, one has the following.

Checkpoint distribution at first adjoint step:	0	30	60	80	94

as shown also in Fig. 3. Hence, the desired resilience distance is taken into account. Assume now, that a breakdown happens directly after the computation of the 57th adjoint step. Because of the chosen distance between to adjoint checkpoints, the last adjoint checkpoint contains the adjoint state 64 and the seven adjoint states 63 to 57 are lost. Therefore, the adjoint computation has to be restarted at the adjoint state 64. However, we face the following difficulty with respect to the available state checkpoints:

Checkpoint distribution at 64th adjoint step:	0	30	60	64	65
Checkpoint distribution at breakdown:	0	30	44	51	56

See also Figs. 4 and 5 for an illustration. Therefore, to recover the seven adjoint steps that were lost because of the breakdown, we cannot just restart resrevolve at the current checkpoint distribution.

To handle this situation, we designed resevolve such that the current checkpointing distribution, the checkpointing distribution that was available when the last adjoint checkpoint was taken, and all information required to restart resrevolve at the last adjoint state available is written to two files when a breakdown occurs. One of the files stores all information required from the data structure cp_schedule; the other one stores information about the state such as the actual values contained in the checkpoints.

```
...
cp_schedule=new Schedule(l,c)
cp_schedule–>set_resilience_distance(distance_resil);
cp_schedule–>set_adjoint_distance(distance_adjoint);
if files exist {
   state–>init(...)
   reconvene_revolve_internals(...) }
do
whatodo = cp_schedule–>next()
   switch(whatodo)
      ...
      case takeshot: store(x, xstore, cp_schedule–>curr_checkpoint)
                     last_checkpoint_stored = cp_schedule–>getcheck()
                     last_state_stored = cp_schedule–>getcapo()
      case firsturn:  eval_J(x, u), init(bar_u, bar_x), adjoint(bar_x, bar_u, x, u)
                     last_adjoint_stored = cp_schedule–>getcapo()
                     store(bar_x, bar_u,bar_xstore,bar_ustore)
      case youturn_with_check: adjoint(bar_x, bar_u, x, u)
                     last_adjoint_stored = cp_schedule–>getcapo()
                     store(bar_x, bar_u,bar_xstore,bar_ustore)
      ...
while(whatodo <> terminate)
...
```

Figure 6. Implementation (resrevolve) of the resilient binomial checkpointing algorithm

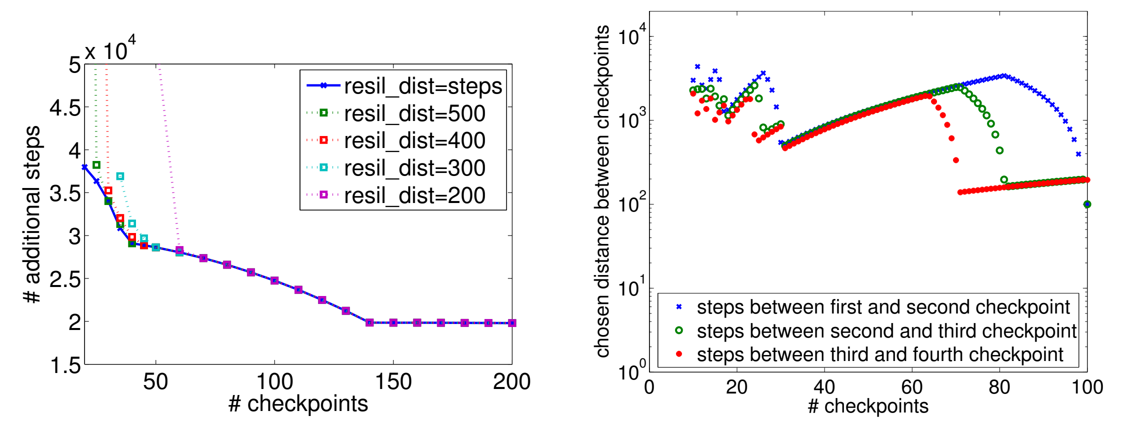
Therefore, when starting an adjoint computation, one first has to check whether these files are available. In this case, one has to reconvene a previous adjoint computation. If the two files are not present, the adjoint computation just starts and the usual initialization of resrevolve is performed. The adjoint computation with resilient binomial checkpointing as illustrated above is implemented as shown in Fig. 6, where only the actions modified compared with Fig. 1 are detailed. As can be seen, in addition to the required initializations only one new action is introduced, namely, the execution of one adjoint step in combination with the storage of the current adjoint state. Hence, if one has already set up the adjoint computation with the original binomial checkpointing, remarkably few changes are needed in order to adapt it for resilience purposes.

4. Analysis of Binomial Checkpointing with Resilience

4.1 *Correctness of Adjoint Computation*

To verify the correctness of the adapted checkpointing approach implemented in resrevolve, we consider the adjoint computation for the following small test case

$$
\begin{array}{rlll}
 & \min J(x,u) \quad \text{with} & & J(x,u) \equiv x_2(1), \\
\text{s.t.} & x_1'(t) = 0.5x_1(t) + u(t), & & x_1(0) = 1 \\
 & x_2'(t) = x_1(t)^2 + 0.5u(t)^2, & & x_2(0) = 0 \\
 & t \in [0,1] & &
\end{array}
$$



(a) Additional time steps needed for adjointing 10,000 time steps

(b) Distance of first four checkpoints chosen by revolve

Figure 7. Complexity results and first three checkpoint distances for 10,000 time steps

adapted from an example proposed by Hager [7]. For this optimization problem, the adjoint can be derived analytically, yielding the following.

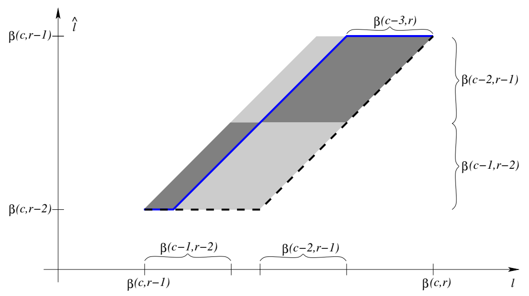
$$\begin{aligned} \lambda_1'(t) &= -0.5\lambda_1(t) - 2 * x_1(t)\lambda_2(t) & \qquad \lambda_1(1) &= 0 \\ \lambda_2'(t) &= 0 & \lambda_2(1) &= 1 \end{aligned}$$

Hence one can verify the correctness of the adjoints computed with resrevolve, that is, including the (repeated) restart using the information stored in the additional files. Indeed, we tested and verified the correctness of the adjoint computation using resrevolve for up to 700,000 time steps and break downs occurring at numerous different places.

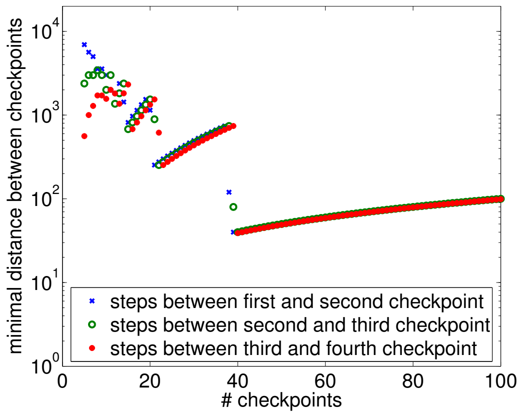
4.2 *Analysis of Runtime Complexity*

An important open point is the temporal complexity of the checkpointing for resilience implemented in resrevolve compared with the one of the optimal binomial checkpointing. As a representative observation, Fig. 7(a) illustrates with a solid line the additional recomputations of time steps needed for 10,000 steps and a varying number of state checkpoints. The additional recomputations needed by the binomial checkpointing approach that incorporates resilience are illustrated with dotted lines for the resilience distances of 200, 300, 400, and 500 steps. Here, we note that the number of checkpoints c and the resilience distance d cannot be chosen completely independently from each other. Because d is the maximal number of time steps between two consecutive checkpoints, it must hold for the forward integration comprising l time steps to be adjointed that $l \leq d \cdot c$. This bound limits the resilience distance from below for a small number of checkpoints, as illustrated in Fig. 7(a). If l is close to the upper bound $d \cdot c$, many recomputations have to be performed since this corresponds to the strategy of complete recomputation for a large part of the forward integration. This explains the high number of additional time steps required for a c, d combination where the product of both values is close to l. On the other hand, we can be see from this example that the resrevolve interferes with the optimality of revolve only if the number of state checkpoints is less than 0.6% of the computed intermediate states.

At first sight this might be a little surprising, especially because the resilience distance



(a) Possible places for the next state checkpoint $\hat{l}$ and two checkpointing strategies



(b) Distances between the first four checkpoints for alternative strategy

Figure 8. Two checkpointing strategies and resulting checkpoint distances for 10 000 time steps

is considerably smaller than the distance of the checkpoints chosen by revolve, as illustrated in Fig. 7(b), for the first four checkpoints to compute the adjoint of 10,000 steps with a varying number of state checkpoints.

To explain the surprisingly good temporal complexity of the binomial checkpointing for resilience implemented in resrevolve, we must examine the strategy to place the next state checkpoint $\hat{l}$. For a given number of time steps l the adjoint of which has to be computed and for a given number of state checkpoints c, let r be the unique integer such that

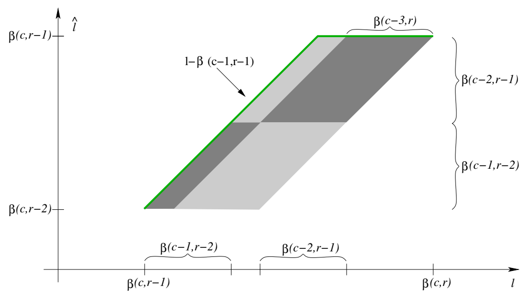
$$\beta(c, r-1) < l \leq \beta(c, r) \equiv \binom{c+r}{r} \tag{3}$$

holds, where we assume $\beta(c, r) = 0$ for $r < 0$. It was shown in [5] that then no time step is executed more than $r + 1$ times. Therefore, the integer r is also called the repetition number. As shown in the same paper, for almost all cases numerous possibilities exist, since using all states $\hat{l}$ fulfilling

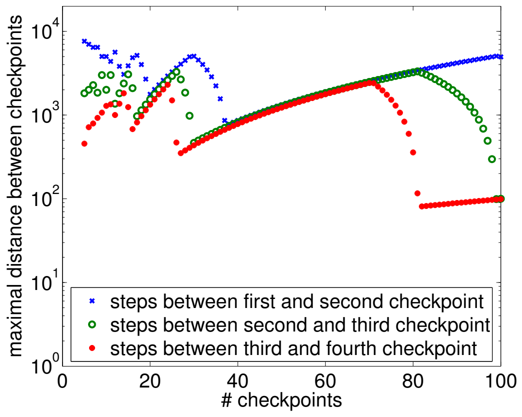
$$\max\{\beta(c, r-2), l - \beta(c-1, r)\} \ \leq\ \hat{l}\ \leq\ \min\{\beta(c, r-1), l - \beta(c-1, r-1)\} \tag{4}$$

as the next state checkpoint leads to the minimization of the number of time steps to be executed. The application of this rule in a recursive way then yields a checkpointing strategy to compute the adjoint with the least possible number of recomputations. The possible choices of $\hat{l}$ are illustrated in Fig. 8(a) as a large, light-gray diamond. In addition to this complexity measure, one can minimize the number of times a state checkpoint is written; see [5, Prop. 2]. The corresponding possible choices of $\hat{l}$ are illustrated by the two small diamonds in dark gray in Fig. 8(a). The approach chosen by revolve takes both criteria into account and is illustrated by the blue line in Fig. 8(a).

We note that the distance between two consecutive state checkpoints has absolutely no influence on the two optimality criteria described above. From a resilience point of view, an obvious alternative strategy for choosing the next state checkpoint minimizing only the number of time steps is illustrated as a dashed black line in Fig. 8(a) and can



(a) Possible places for the next state checkpoint $\hat{l}$ when maximizing checkpointing distances



(b) Resulting distances of the first four checkpoints

Figure 9. Third checkpointing strategy and resulting checkpoint distances for 10,000 time steps

be described by

$$\hat{l} = \max\{\beta(c, r-2), l - \beta(c-1, r)\} \, .$$

For the small example in Section 3 (i.e., $l = 100$ and $c = 5$), one obtains for the computation of the 64th adjoint state the following state checkpoint distribution.

$$0 \quad 30 \quad 45 \quad 55 \quad 61$$

Hence, this checkpoint distribution fits the resilience distance of 30. For the larger example considered in this section, the effects on the checkpointing distribution are illustrated in Fig. 8(b). As can be seen, the number of time steps between two consecutive state checkpoints can be dramatically reduced.

One can observe in the Figs. 7(b) and 8(b) that the distance between two consecutive state checkpoints decreases monotonically in most cases but unfortunately not always making it difficult to explain the surprisingly good complexity result for the binomial checkpointing with resilience. For this reason, we examine the counterintuitive approach of maximizing the distances between two consecutive state checkpoints by choosing

$$\hat{l} = \min\{\beta(c, r-1), l - \beta(c-1, r-1)\}$$

as illustrated also in Fig. 9(a). The resulting checkpointing distances are shown in Fig. 9(b) and decrease monotonically. One can also prove, that this property always holds. For this purpose, we first need the following result.

Lemma 4.1 *The function $f : \mathbb{N} \mapsto \mathbb{N}, f(c) = \beta(c, r)$, is monotonically increasing for any given fixed value of $r \in \mathbb{N} \cup \{0\}$.*

Proof. One has

$$\beta(c-1, r) \leq \beta(c, r) \;\Leftrightarrow\; \frac{(c+r-1)!}{r!(c-1)!} \leq \frac{(c+r)!}{r!\,c!} \;\Leftrightarrow\; 1 \leq \frac{c+r}{c},$$

which holds if $r \geq 0$. ■

Now, we prove the main theoretical result of this section:

THEOREM 4.2 *For a given time step number $l > 1$ and a state checkpoint number $c \geq 2$, the checkpointing strategy*

$$\hat{l} = \begin{cases} \beta(c, r-1) & \text{if } \beta(c, r-1) + \beta(c-1, r-1) \leq l \leq \beta(c, r) & (5) \\ l - \beta(c-1, r-1) & \text{if } \beta(c, r-1) \leq l \leq \beta(c, r-1) + \beta(c-1, r-1) & (6) \end{cases}$$

yields an optimal (i.e., time-minimal), checkpointing schedule, where the distances between the checkpoints are monotonically decreasing.

Proof. The time minimality follows from Eq. (4) with [5, Prop. 1]. To prove the monotonicity, we have to examine all possible cases. For this purpose, assume first that $\hat{l} = \beta(c, r-1)$, and let $\tilde{l}$ denote the place of the subsequent checkpoint. Now, we have to distinguish two situations. If $\tilde{l} = \beta(c-1, r-1)$, then one obtains

$$\begin{aligned} \hat{l} = \beta(c, r-1) = \binom{c+r-1}{r-1} = \binom{c+r-2}{r-1} + \underbrace{\binom{c+r-1}{r-1}}_{\geq 0} \\ \geq \binom{c+r-2}{r-1} = \beta(c-1, r-1) = \tilde{l}, \end{aligned}$$

proving the assertion for this case. If $\tilde{l} = l - \hat{l} - \beta(c-2, r-1)$ holds, we can exploit that

$$\beta(c-2, r) \leq l - \hat{l} \leq \beta(c-1, r-1) + \beta(c-2, r-1)$$

yielding

$$\begin{aligned} \tilde{l} = l - \hat{l} - \beta(c-2, r-1) &\leq \beta(c-1, r-1) + \beta(c-2, r-1) - \beta(c-2, r-1) \\ = \beta(c-1, r-1) &\leq \beta(c, r-1) = \hat{l}, \end{aligned}$$

where the last inequality follows from Lemma 4.1 for $r \geq 1$. Hence, the assertion is true also in this case.

Second, assume that $\hat{l} = l - \beta(c-1, r-1)$. Once more, we have to distinguish two situations. For the case $\tilde{l} = \beta(c-1, r-1)$, we can exploit that

$$\beta(c-1, r-1) + \beta(c-2, r-1) \leq l - \tilde{l} \leq \beta(c, r-1)$$

yielding

$$\hat{l} - \tilde{l} = l - \beta(c-1, r-1) - \tilde{l} \geq \beta(c-1, r-1) + \beta(c-2, r-1) - \beta(c-1, r-1) \geq 0,$$

and therefore the desired relation $\hat{l} \geq \tilde{l}$. Finally, we have to consider the case $\tilde{l} = l - \hat{l} -$

$\beta(c-2, r-1)$. Then, it follows with

$$\begin{aligned}\hat{l} &= l - \beta(c-1, r-1) \geq \beta(c, r-1) - \beta(c-1, r-1) = \binom{c+r-1}{r-1} - \binom{c+r-2}{r-1} \\ &= \binom{c+r-2}{r-2} + \binom{c+r-2}{r-1} - \binom{c+r-2}{r-1} = \binom{c+r-2}{r-2} = \beta(c, r-2)\end{aligned}$$

that

$$\begin{aligned}\hat{l} - \tilde{l} &= l - \beta(c-1, r-1) - \Big(l - \hat{l} - \beta(c-2, r-1)\Big) \\ &= \hat{l} + \beta(c-2, r-1) - \beta(c-1, r-1) \\ &\geq \beta(c, r-2) + \beta(c-2, r-1) - \beta(c-1, r-1) \\ &= \binom{c+r-2}{r-2} + \binom{c+r-3}{r-1} - \binom{c+r-2}{r-1} = \binom{c+r-3}{r-3} \geq 0,\end{aligned}$$

proving the assertion also for this last case. ■

An immediate consequence of this strategy for choosing the next checkpoint is that the distance between the last state checkpoint set during the first forward integration and the state l is minimized. This observation will be used later in the numerical comparison of the different approaches. Based on this result, to determine the minimal checkpoint distance that can be achieved, one has to check only the corner cases. These corner cases occur either when one switches from case 1 to case 2 with the same value of r or when one switches from case 2 to case 1 and the value of r increases by one. However, all corner cases have to be examined because there is no additional monotonicity as illustrated by the next lemma.

Lemma 4.3 *There exist $l_i, c_i, r_i, i = 1, \ldots, 4$, such that*

$$\begin{array}{rclcrcl}\beta(c_1, r-1) &>& l_1 - \beta(c_1+1, r_1-1), && l_3 - \beta(c_3-1, r) &>& \beta(c_3+1, r_3-1), \\ \beta(c_2, r-1) &<& l_2 - \beta(c_2+1, r_2-1), && l_4 - \beta(c_4-1, r) &<& \beta(c_4+1, r_4-1)\end{array}$$

Proof. Choosing $l_1 = 20000, c_1 = 21, r_1 = 5$ yields $\beta(c_1, r_1-1) = 10625 > 9374 = l_1 - \beta(c_1+1, r_1-1)$. For $l_2 = 10000, c_2 = 25, r_2 = 4$, one has $\beta(c_2, r_2-1) = 4960 < 5040 = l_2 - \beta(c_2+1, r_2-1)$. Setting $l_3 = 10000, c_3 = 139, r_3 = 3$, it follows that $l_3 - \beta(c_3-1, r_3) = 270 > 141 = \beta(c_3+1, r_3-1)$. For $l_4 = 15000, c_4 = 22, r_4 = 5$, one obtains $l_4 - \beta(c_4-1, r_4) = 2350 < 2600 = \beta(c_4+1, r_4-1)$. ■

From a practical point of view, one can now compute the corner cases for a given number l of time steps to determine the number of state checkpoints required to achieve a desired resilience distance that does not interfere with the runtime optimality of the binomial checkpointing together with an acceptable repetition number, that is, time to required to compute the adjoint information.

To illustrate the development of the achievable resilience distance d, we list some examples in Table 1 referring to one, two, and five years of physical time, as mentioned in Section 1.

Table 1. Resilience distances d for different values of l and r

l	350 000	700 000	1 750 000
r	4	4	4
c	52	62	78
d	26 235	43 680	86 260
r	3	3	3
c	127	160	217
d	8256	13 041	23 871
r	2	2	2
c	836	1182	1870
d	837	1183	1871

5. Other Checkpointing Approaches for Resilience and Comparisons

In this section we describe other approaches that can be used for restarting checkpointing and adjoint checkpointing. For this purpose, one has to keep in mind the application that motivated the research presented in this paper. That is, there is a fixed time limit T when the execution of the adjoint computation must be interrupted the latest as explained in Sec. 1. When t_a denotes the time needed for the execution of one adjoint step, the inequality

$$(r + t_a)d \leq T,$$

with r being again the repetition number, ensures that the adjointing of no more that d time steps is possible within this time limit. Hence, if one chooses for given l, d, and t_a, the number of checkpoints such that the last inequality holds, the binomial checkpointing with resilience can be used to compute the required adjoint information.

Nevertheless, we compare the proposed binomial checkpointing with resilience also with other checkpointing appoaches that aim at resilience. For this purpose, we considered again one, two, and five years of physical time yielding the number of time steps as shown in Table 1 and breakdowns occuring at different places in the first forward integration and during the actual adjoint computation.

5.1 *Continuous Checkpointing*

In continuous checkpointing, a simulation containing l time steps is allowed to use c persistent checkpoints for binomial checkpointing. Up to the very end of the first forward integration, at least one state checkpoint is free. This free state checkpoint is overwritten whenever possible until it must serve as a state checkpoint for the binomial checkpointing (see Fig. 10). As long as the time to write a state checkpoint is practically negligible, this approach provides immediate recovery until the storage for the last free state checkpoint must be used for the binomial checkpointing.

Hence, this checkpointing strategy does not require any additional means to cover resilience. Therefore, it would be the ideal case. However, usually it is not possible to store all state checkpoints in persistent memory and to retrieve them with negligible time when needed. Therefore, we just sketch this possibility but do not analyse it further.

From a theoretical point of view this ideal checkpointing strategy for resilience is

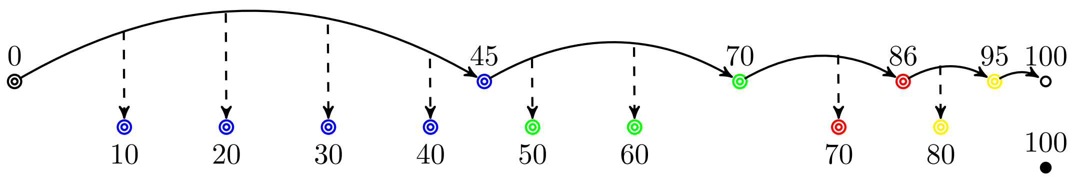


Figure 10. Checkpoint distribution using regular revolve with continuous checkpoints. Checkpoints with the same color are to the same location.

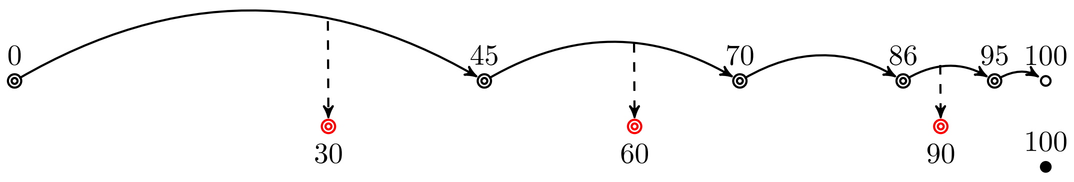


Figure 11. Checkpoint distribution at first adjoint step using regular revolve with an additional resilience checkpoint. Checkpoints with the same color are to the same location.

interesting since it contradicts the so-called checkpointing stability that can be observed in the optimal checkpointing strategies.

5.2 *Single Recovery Checkpoint*

An implementable approach is given by the single recovery checkpoint approach. Having a total number of c state checkpoints available, then $c-1$ checkpoints are used for the binomial checkpointing of the state (see Fig. 11). An additional checkpoint is used to store the state of the computation at intervals of resilience distance d irrespective of when the binomial checkpointing method stores its state checkpoints.

Hence if no breakdown occurs, the minimal number of time steps executed by the binomial checkpointing with $c-1$ state checkpoints for adjointing l time steps equals

$$lr - \beta(c, \tilde{r} - 1)$$

with $\tilde{r}$ such that $\beta(c-1, \tilde{r}-1) < l \leq \beta(c-1, \tilde{r})$. Since here one state checkpoint less is used for the adjoint computation this number must be larger than the minimal number of time steps required for the adjoint computation if all c state checkpoints are used for the adjoint computation and the values of l, c, r, and d are such that the resilience distance does not interfere with the optimality of the binomial checkpointing. Hence, if no breakdown occurs at all, the binomial checkpointing with resilience leads to a smaller runtime for the adjoint computation. If a breakdown occurs the cost to restart the adjoint computation is depends on the place of the breakdown and the state checkpoints stored. Therefore, the number of time steps required for the restart is difficult to determine in a general way.

For the numerical test that we performed the binomial checkpointing with resilience yields in more than 75 % of the tested configurations a smaller number of time steps to be executed than the single recovery checkpoint approach. Furthermore, the single recovery checkpoint approach does not take a given time limit T into account to prepare for a suitable restart making the application of this strategy for our target application difficult.

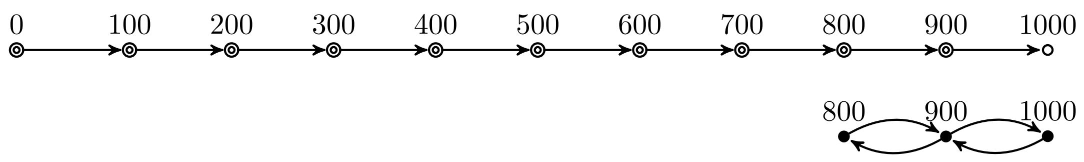


Figure 12. Checkpoint distribution using two-level checkpointing. First resilient, state checkpoints are stored at coarse granularity by running the unmodified function. The lower level uses binomial checkpointing without resilience considerations.

5.3 *Two-Level Checkpointing*

In two-level checkpointing, a simulation containing l time steps can be divided into l_1 outer intervals each of which contains l_2 time steps, such that $l = l_1 * l_2$. In the first forward integration, the outer level is checkpointed at regular intervals for resilience. In the adjoint sweep, at the start of each l_{1_i} time step, the state checkpoint written at the end of the $l_{1_{i-1}}$ time step is read (see Fig. 12). To compute the adjoint of the ${l_{1_i}}^{th}$ step, binomial checkpointing is used with a repetition number r_2 such that

$$\beta(c_2, r_2 - 1) < l_2 \leq \beta(c_2, r_2)\ . \tag{7}$$

Once the adjoint is computed, the adjoint is checkpointed for resilience. If a failure occurs in the forward integration, the last outer level state checkpoint is used for recovery and at most $2 * l_2$ forward steps need to be recomputed. If a failure occurs in the adjoint integration, then the appropriate forward state checkpoint and adjoint checkpoint can be used for recovery.

With respect to the resulting time complexity, this approach suffers from the equally distributed state checkpoints at the outer level. If no breakdown occurs, the resulting number of time steps to be executed is given by

$$l_1 * (l_2 r_2 - \beta(c_2, r_2 - 1))\ ,$$

which again must be larger than the temporal complexity of the binomial checkpointing when resilience does not interfere with optimality. For all test configurations that we considered this approach leads also to a significantly higher temporal complexity that the binomial checkpointing with resilience when breakdowns occur.

6. Conclusions and Future Work

We have presented a resilient binomial checkpointing algorithm that supports the restart of the adjoint computation after a failure of the computing system. The modified algorithm maintains the optimality of binomial checkpointing while limiting the maximum distance between successive state and adjoint checkpoints. The required changes were integrated in the software package revolve for binomial checkpointing and will be made available on the web site of revolve as stated at the tool list web site on www.autodiff.org.

From a theoretical perspective, it would also be interesing to introduce not a fixed distance for the adjoint checkpoints but to coordinate the storing of an adjoint checkpoint with the progress of the adjoint computation. This would reduce the number of time steps executed to reconvene the adjoint computation. Future work be will dedicated to this subject.

We plan to apply the resilient binomial checkpointing algorithm to compute the adjoint of the MITgcm. It has previously been differentiated by OpenAD [10] using the original binomial checkpointing algorithm. This requires us to examine the additional data that must be checkpointed in order to support restart of the application. This includes OpenAD's `tape` data structures that are used to hold intermediate values as well as global data structures that are used outside the time-stepping loop. We plan to use a modified version of OpenAD's template mechanism for `revolve` to support the restart of the computation.

References

[1] A. Adcroft, C. Hill, and J. Marshall, *Representation of topography by shaved cells in a height coordinate ocean model*, Monthly Weather Review 125 (1997), pp. 2293–2315.

[2] A. Adcroft, J.M. Campin, C. Hill, and J. Marshall, *Implementation of an atmosphere ocean general circulation model on the expanded spherical cube*, Monthly Weather Review 132 (2004), pp. 2845–2863.

[3] G. Aupy, J. Herrmann, P. Hovland, and Y. Robert, *Optimal multi-stage algorithm for adjoint computation*, SIAM Journal on Scientific Computing (2016), to appear.

[4] A. Griewank, *Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation*, Optimization Methods and Software 1 (1992), pp. 35–54.

[5] A. Griewank and A. Walther, *Algorithm 799: Revolve: An implementation of checkpoint for the reverse or adjoint mode of computational differentiation*, ACM Transactions on Mathematical Software 26 (2000), pp. 19–45.

[6] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, SIAM, 2008.

[7] W. Hager, *Runge-Kutta methods in optimal control and the transformed adjoint system.*, Numerische Mathematik 87 (2000), pp. 247–282.

[8] M. Schanen, O. Marin, and H.Z. amd M. Anitescu, *Asynchronous two-level checkpointing scheme for large-scale adjoints in the spectral-element solver Nek5000*, Tech. Rep. ANL/MCS-P5422-1015, Argonne National Laboratory, 2015.

[9] P. Stumm and A. Walther, *Multi-stage approaches for optimal offline checkpointing.*, SIAM Journal of Scientific Computing 31 (2009), pp. 1946–1967.

[10] J. Utke, U.N.M. Fagan, N. Tallent, M. Strout, P. Heimbach, C. Hill, and C. Wunsch, *OpenAD/F: A modular open-source tool for automatic differentiation of Fortran codes*, ACM Transactions on Mathematical Software 34 (2008), pp. 18:1–18:36.

Acknowledgments

We thank Paul Hovland, Laurent Hascoët, Dan Jones and Patrick Heimbach for their suggestions.

Funding

This work was funded in part by a grant from DAAD Project Based Personnel Exchange Programme and by support from the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.